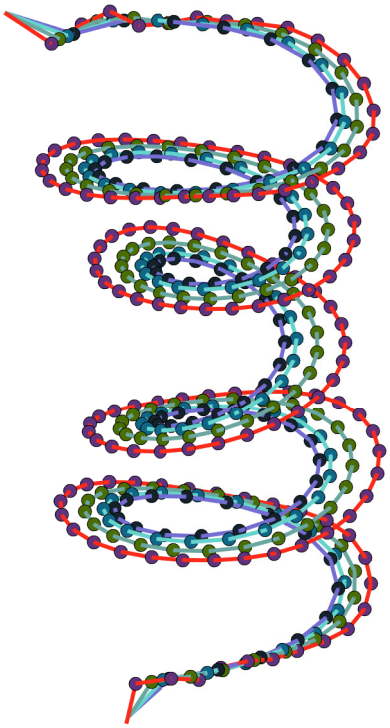
PCA Manual

OR

How to Succeed in Rescaling of Polymers

Anna Sinelnikova



Contents

Introduction

PCA project consists from:

1. Independent PCA library in **C++11**.
2. Three example applications in **C++11**.
3. **MATLAB** script for plotting one or several chains which are obteined during scaling procedure.
4. Another **MATLAB** script for making movies of scaling process, where one scaling step is one frame.
5. Small script for **GNUplot** for plotting the figure of dependency of total angle on scaling step.
6. Some chains of proteins from Protein Data Bank (PDB) for quick start.
7. Library documentation for developers, which was generated automatically in "doxygen".

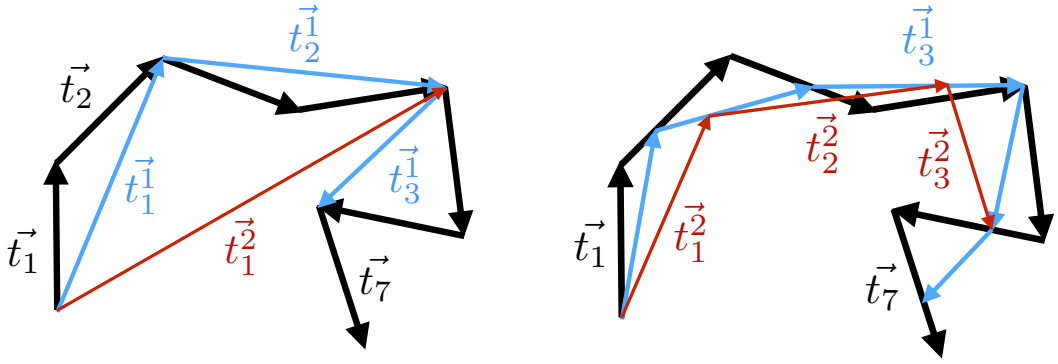
The program unfortunately does not have any graphical user interface. So you can work with it only from the terminal. However the only thing you should know is how to change the directory there. All the rest you can do with your favourite file managers and text editor.

1 Algorithm

1.1 Theory

Let us assume we have a chain of solid rigid segments. All we need to describe this kind of system is coordinates of all sites of the chain. Now we want to do some kind or rescaling of the system. The simpliest idea is to connect every second site with new link. It is illustrated in Figure 1 (a) with blue lines. If $\vec{t_1}, \vec{t_2}, ...$ are vectors along links of original chain, then after the first step of rescaling procedure we get $\vec{t_1^1}, \vec{t_2^1}, ...$ This new blue chain also can be rescaled in the same way, thus we have the second step of the procedure: red line in Figure 1 (a) $\vec{t_1^2}, \vec{t_2^2}, ...$ For long enough chains we can have more than 3 iteration steps. However, the length of the original chain reduces quite quick, by 2^p, where p is number of iteration step.

Let s denote number of old monomers in a new one. Then in the previous case $s = 2$. What if we take not integer s? This situation is presented in Figure 1 (b). We do quite similar to the previous case thing, but instead of connecting two old links in a new one, we connect four thirds. More strictly for the first iteration step:



(a) $s = 2$. (b) $s = \frac{4}{3}$.

Figure 1: Illustration of two first steps of scaling procedure for various scaling parameter s.

$$\vec{t}_1^{\,1} = \vec{t}_1 + \frac{1}{3}\vec{t}_2$$
$$\vec{t}_2^{\,1} = \frac{2}{3}\vec{t}_2 + \frac{2}{3}\vec{t}_3$$
$$\vec{t}_3^{\,1} = \frac{1}{3}\vec{t}_3 + \vec{t}_4$$

For the second iteration step we do the same but instead of the original chain we take the chain we got from the previous step.

1.2 General algorithm

Let us consider the general algorithm for any $s \in (1, +\infty)$,

We can express s as a rational fraction:

$$s = I + \frac{n}{d}, \quad \text{where} \quad n \in \mathbb{N}; \quad I, d \in \mathbb{N}\backslash 0 \tag{1}$$

Despite that the idea of that algorithm is not complicated, it is not possible to write it mathematically in a short way (like for the previous case), because of complicated indexation during scaling procedure. So let's look at the working algorithm instead, which is performed by function **`scaling`** from class **`PolymerScaling`**. This function is the core of the program, all other functions from the same class call **`scaling`** finally.

The flowchart of the algorithm is presented in Figure 2. There we used together with (1) this expansion of scaling parameter:

$$s = \text{intS} + \text{fracS} \tag{2}$$

"k%d" means that you should take the integer part of quotient, according C notation.

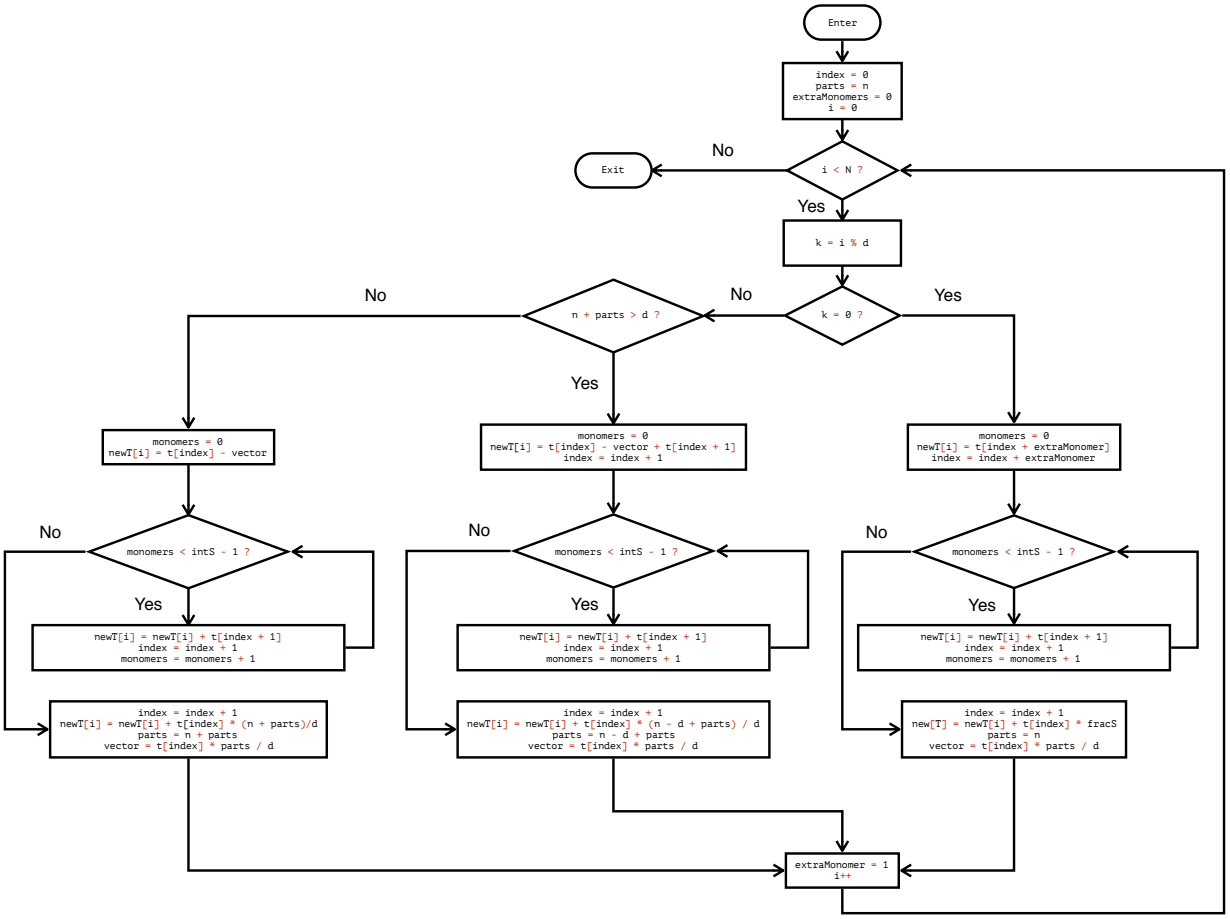


Figure 2: The flowchart of function **`PolymerScaling::scaling(...)`**

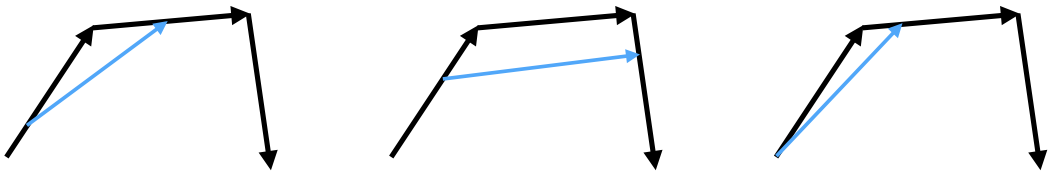
There are three cases in the algorithm. They are schematicaly shown in Figure 3, where black chain is the old one, and blue is new. The right branch (Figure 3c) is the situation when new link starts from some old site. It means that some old site concisest with new one. Two other cases describe the opposite situation: new site divides old link into two parts. The difference between left branch and the middle one can be easily seen in figures 3b and 3c.

2 Quick Start

Here is the instruction how one can quick run the program. The program does not have graphical interface, so all the work should be done from the terminal.

2.1 Compilation the library

First of all download PCA project to your computer from https://github.com/Anny-Moon/PCA. You can do it manually on the website or you can use **`git`**:



(a) branch 1 (b) branch 2 (c) branch 3

Figure 3: to flowchart

```
$ git clone https://github.com/Anny-Moon/PCA
```

The next step is to compile the PCA library on your computer (you are required to have **gcc** compiler). Go to folder **/PCA_lib** in the main folder with project and do **make**:

```
$ cd PCA/PCA_lib
PCA/PCA_lib$ make
```

The compiler will show you which files it handled. And this line will show that library was successfully built:

```
Generating static library... libpca.a
```

Now we want to make the library accessible from any folder on you computer. It will allow you to create applications and do not care about paths. Go back to root folder and start initialization session:

```
PCA/PCA_lib$ cd ../
PCA$ source init_session.sh
```

Important: initialization session should be done every time you start new terminal.

2.2 Applications

There are three things you can do:

1. **scaling**: Scaling procedure without calculating any observables.
2. **totalAngle**: Scaling procedure with calculating total angle for fixed parameter k (see eq. 10). If you have several chains in your file (see sec. 7.1) then it will calculate average total angle and error (see sec. 6.2).
3. **various_k**: Scaling procedure with calculating total angle for broad range of k. If you have several chains in your then it will calculate average total angle and error.

Copy application you want to use somewhere on your computer. Then go to this folder and run make:

```
myApplication$ make
```

Since you see this:

```
Generating executable file.. pca
```

you are ready to start calculations.

Choose which polymer from folder **data** in the project's root you want to analyze and pass the name as the first argument for the program like this:

```
PCA$ ./pca 5dn7
```

It will run function **observableVSscalingStep** for observable **TotalAngle** for chain 5dn7 with writing all possible side files (see sec. 6.1). All the files will be put in folder **/results**.

For the program number two you can also pass parameter k as the second argument (by default $k = 10$). For example:

```
PCA$ ./pca 5dn7 50
```

for $k = 50$.

Tip: run any program without arguments and it will tell you about itself and what it expects you to pass:

```
PCA$ ./pca
```

2.3 Handle the Results

If you have **GNUplot** you can immediately plot dependency of total angle on scaling step.

```
PCA$ gnuplot -e "polymerName='5dn7'" plotTotalAngle.gnu
```

3 Class Polymer

This class is the main class of the program. It stores the information about the model. On of the possible way of calling the constructor is the following:

```
Polymer polymer(inputFileName, linesInBlock, blockNumber);
```

Constructor of **class Polymer** takes name of the file with coordinates of atom (see sec. **??**) as the first argument. The rest is not needed. You can pass both the the 2nd and the 3rd arguments, only one, or none. The meaning of them is thus:

2. "linesInBlock" — number of atoms in your model. By default this argument is equal to 0. This forces constructor to count number of atoms itself. This is preferable situation. So pass the second argument as 0 all the time if you are not restricted in time.

3. "blockNumber" — number of chain in the file. By default the value is equal 1. So it is the first (and could be the only one) data block in your file.

Tip: if you do not know how many blocks you have in the file, but want to know then you can call either function **`void showNumberOfLinesInBlocks(char* fileName)`** or **`int countBlocks (char* fileName)`** (see sec. 8.3).

4 Scaling Parameter

Scaling parameter s denotes how many old monomers will be in a new one (see sec. 1). It is a **`double`** number which is **grater than 1**. It is stored like this:

$$s = I + \frac{n}{d}, \quad n \in \mathbb{N}; \;\; b, I \in \mathbb{N}\backslash 0 \tag{3}$$

in a class **`ScalingParam`** which is a member of class **`PolymerScaling`**.

There are two contractors (plus copy constructor) for scaling parameter:

```
ScalingParam(double s_in);
ScalingParam(int nemerator_in, int denominator_in, int intPart_in=1);
```

So both these lines:

```
PolymerScaling::ScalingParam s(1.25);
PolymerScaling::ScalingParam s(1, 4);
```

will create scaling parameter $s = 1.25$.

Important: by default all scaling functions from sec. 6 realise floating scaling parameter mode (see the following section). Write in your main:

```
PolymerScaling::enableFloatingScalingParam = false;
```

to fix scaling parameter during all scaling procedures.

4.1 Floating Scaling Parameter

If **`PolymerScaling::enableFloatingScalingParam`** set to **`true`** (by default), then the following mode will be activated.

At each iteration step the program will find the closest scaling parameter to the desired one to provide exact integer number of monomers in the new chain. It means that the end of the chain will be strictly fixed[1] and we will not loose it like in figure 1.

[1] In practice it is fixed up to numerical errors because of finite number of significant digits.

Let's consider an example. Take chain with 153 monomer, $N = 153$. You want to take $s = 1.3$. The algorithm we described above is performed in function **`findNewScalingParam(N)`** and looks like following.

1. Round the number of monomers to the closest integer.

$$N^{new} = \frac{N}{s} = \frac{153}{1.3} = 117.69 \Rightarrow 118 \tag{4}$$

2. Found new scaling parameter

$$s^{new}\frac{N}{N^{new}} = \frac{153}{118} = 1.2966 \tag{5}$$

You see, instead of desired $s = 1.3$ at the first step of rescaling $s = 1.2966$. The next step **`findNewScalingParam(N)`** will be called again. And since the chain has 118 monomers now, $s^{new} = 1.2967$. The next scaling step s again will be equal to 1.3. This procedure will be continued until we get chain with only 3 monomers, so only one iteration step left: $s^{new} = 1.5$; $N^{new} = 2$.

4.2 Some Programming Detales

You may want to read this section if you have the first problem from the error list (sec. 9).

Let's say you want scaling parameter to be $s = 1.3$. But for performing geometrical transformation (what process of scaling is) the program should convert this decimal fraction to rational one. And it is exactly what function **`void findEverythingFromS()`** (which is a private member of class **`ScalingParam`**) does. There are 3 steps:

1. The function splits integer and fraction part of the value. So $s = 1.3 = 1 + 0.3$.

2. It takes fractional part and do this:

$$0.3 \Rightarrow \frac{0.3 * 10000}{10000} = \frac{3000}{10000} \tag{6}$$

3. The function starts the loop to abbreviate by cancel the fraction

$$\frac{3000}{10000} \Rightarrow \frac{6}{20} \Rightarrow \frac{3}{10} \tag{7}$$

So finaly we have:

$$s = 1.3 = 1\frac{3}{10} \tag{8}$$

The value 10000 which we use in step 2 is actually stored in **`double ACCURACY`**. There are two reasons why you might want to change it: you got the error massage or for some reasons you want to have more than 4 decimal digits.

However, we are stronly recommend you **not** to increase **ACCURACY** a lot. Because the program use scheme of scaling with "floating" scaling parameter. What means that all these three steps are happend at each step of scaling procedure. And there are another loop in step 3 for finding common divisor which can be also very time consuming.

Anyway, there are two functions for set and get current **ACCURACY** value. You can call them like this:

```
PolymerScaling::ScalingParam::setAccuracy(1e+07);
currentAccuracy = PolymerScaling::ScalingParam::getAccuracy();
```

5 Observables for Scaling Functions

When you perform scaling procedure you can choose one of four values to be calculated. All of them are members of enumeration class **enum class Observable** which is a member of the class **PolymerScaling**. So there are four members.

5.1 scalingParameter

5.2 totalAngle

This name corresponds to function **totalAngle** in class **PolymerObservable**. If we define $\vec{t}_i$ as a tangent vector, i.e.

$$\vec{t}_i = \vec{r}_{i+1} - \vec{r}_i \tag{9}$$

where $\vec{r}_i, \vec{r}_{i+1}$ are radius vectors of corresponding atoms, then total angle is

$$\theta = \sum_{\substack{i=1 \\ j=i+k}}^{N} \frac{(\vec{t}_i \cdot \vec{t}_j)}{|\vec{t}_i||\vec{t}_j|}, \tag{10}$$

N is number of monomers and k is fixed parameter. For getting physical result k should be no less than 10.

5.3 radiusOfGyration

This name corresponds to function **radiusOfGyration** in class **PolymerObservable**. It calculate radius of gyration from coordinates of chain's sites as:

$$R_{gyr} = \sqrt{\frac{1}{2(N+1)^2} \sum_{\substack{i=1 \\ j=1}}^{N+1} (\vec{r}_i - \vec{r}_j)^2}, \tag{11}$$

where $\vec{r}_i, \vec{r}_j$ radius vectors of sites i and j and $N+1$ is a total number of sites (because we denoted N as number of links).

5.4 averageMonomersLength

The conventional mean value of monomer length along the chain:

$$\langle l \rangle = \frac{1}{N} \sum_{i=1}^{N} l_i \tag{12}$$

where N is number of backbones in the polymer.

6 Scaling Functions

In all scaling function we use "floating" scaling parameter. It means that scaling parameter will be recalculated at each scaling step in order to fix the first and the last sites.

6.1 observable VS scaling Steps

The function writes main file with dependency of some value on number of scaling step. It is also possible to write some side files.

All file formats you can find in section 7.2

One of possible call of the function is the following:

```
1 PolymerScaling::observableVSscalingSteps(
2         PolymerScaling::Observable::totalAngle,
3         polymer,
4         sp, resultFile,
5         confFile, nFile, sFile);
```

Let's look at all 7 arguments closer:

1. The first argument is name of observable which you want to measure during scaling process. It can be any observable from **`enum class Observabe`** (see sec. 5). In the example we use total angle.
2. "polymer" — the constant reference to our polymer or **`const Polymer&`**.
3. "sp" — the constant reference to scaling parameter which be taken as etalon scaling parameter or **`const ScalingParam&`** (see sec. **??**).
4. "resultFile" — name of the file i.e **`char*`** where you want to write the result.

The arguments in the 5th line are not necessary. These are three side pointers to files. You can pass some of them to the function or pass none.

5. "confFile" — name of the file for polymers which are obtained at each scaling step. The first block in file is the initial chain [2]. You need this file if you want to track the evaluation of polymer chain during the scaling procedure.

6. "nFile" — name of the file where lengths of chain can be written.

7. "sFile" — name of the file where you want to write values of scaling parameter at each step of the procedure.

6.2 observable VS scaling Steps with Statistics

This function do almost the same as the previous one, but can handle the statistics. It means that if you have set of several chains in your input file then the function will perform the scaling procedure for all the chains. Observable at each step will be calculated as average:

$$\langle x \rangle = \frac{1}{M} \sum_{i=1}^{M} x_i \tag{13}$$

where M is number of chains in statistics. There will be extra column for error of the value which is calculated as standard deviation of the mean:

$$\Delta x = \sqrt{\frac{1}{M^2} \sum_{i=1}^{M} (x_i - \langle x \rangle)^2} \tag{14}$$

Possible call for the function is this:

```
1 PolymerScaling::observableVSscalingStepsWithStatistics(
2         PolymerScaling::Observable::totalAngle,
3         dataFile, statistics,
4         sp, resultFile,
5         confFile, nFile, sFile);
```

It is very similar to the previous function but there is a difference in the 3rd line. You do not need to create object of **`class Polymer`**, but need to pass the name of the file with polymer chains. So the new arguments are:

2. "dataFile" — name of input file with your chains.

3. "statistics" — number of chains in the previous file. This number will go for N in equations 13 ans 14.

[2] The original configuration is parallel shifted in point (0, 0, 0). So if in you input file the first atom was not in the origin of coordinates then coordinates of all atoms will differ from your input. Do not panic, it is still the same chain. Parallel shifting does not change anything.

The rest arguments are the same as for **observableVSscalingSteps**. If you pass the names of side files then the information which correspond only to the first polymer in your input file will be written there.

Important: in input datafile **all** blocks should be of equal size. It means that the length of polymers in statistics is the same. Otherwise averaging has no sense.

Tip: if you are not sure that all your blocks in file are of equal size then call function **void showNumberOfLinesInBlocks(char* fileName)** (see sec. 8.3).

Tip: pass 0 for statistics, then the program will calculate it automatically.

6.3 scaling Loop

Another very useful scaling function is **scalingLoop**. It is not used by the rest of the program, but it gives a lot of freedom to user. Roughly speaking, the function takes your polymer, makes as many scaling steps as you want and replace your polymer with the chain from last scaling step. It is also possible to ask function to write files with coordinates, scaling parameters and numbers of monomers (see sec. 7.2). But unlike previous functions here you should pass already opened files which are available for writing. So it is possible to run this function in a loop.

One of the possible call is the following:

```
PolymerScaling::scalingLoop(
        &polymerPointer,  sp,
        confFp, numMonomersFp, scalingParamFp,
        loopSteps)
```

1. "&polymerPointer" — reference to pointer to your polymer. The function expects you to pass **Polymer** polymer**.
2. "sp" — constant reference to etalon scaling parameter, as in previous functions (see sec. ??).

7 Files Formats

7.1 Input file

All input files should be placed in **/data** in the root of the program. The extantion of the file is **.dat**. Both these settings can be easily changed in **main.cpp**.

Give the name of your file without extantion as an argument when you running the program. Then the program will add folder name and extantion and pass this string to **Polymer** constructor:

```
sprintf(str,"data/xyz_%s.dat",p[1]);
Polymer polymer(str);
```

The format of **.dat** file should be the following:

```
<polymer1_atom1_x>  <polymer1_atom1_y>  <polymer1_atom1_y>
<polymer1_atom2_x>  <polymer1_atom2_y>  <polymer1_atom2_y>
        //other atoms
<polymer1_atom_N_x> <polymer1_atom_N_y> <polymer1_atom_N_y>

<polyme2_atom1_x>   <polymer2_atom1_y>  <polymer2_atom1_y>
<polymer2_atom2_x>  <polymer2_atom2_y>  <polymer2_atom2_y>
        //other atoms
<polymer2_atom_M_x> <polymer2_atom_M_y> <polymer2_atom_M_y>

//other blocks
```

File stores coordinates of atoms. Each line is x, y, z coordinates of one atom which can be separated with **space(s)** or **tab(s)**. New line is the coordinates of the next atom. Empty line separate one polymer chain from another one.

Important: line which starts with unprintable character (**space** or **tab**) counts like empty! That is why data lines cannot start with unprintable character.

And the end of the file you might or might not have a new line, it does not matter.

7.2 Output files

There are several different output files for **Scaling** functions. The first line (or data block) in all files corresponds to the original chain i. e. 0-step of iteration process. All files has **.dat** extantion and goes to **/results** folder in the root of the program by default. As with input files one can change it in **main.cpp**.

7.2.1 Files with results

The first column is a number of scaling step. The second is the **Observable**. If you called **Scaling** function which works with statistics then you will have the third column for errors.

7.2.2 Configurations

The format is the same as in input file with configurations. Each data block contains the coordinates of one polymer chain. Blocks are separated by two empty lines[3].

7.2.3 Number of monomers

Here is just a list of number of backbones for each scaling step.

[3]Two empty lines correspond to blocks separator in **GNUplot**. So you can use **plot <file> index <number of block>** there.

7.2.4 Scaling parameter

In this file one can find the list of scaling parameters for each scaling step. The first line corresponds to the scaling parameter which you passed to **Scaling** functions.

8 Nice Features

8.1 Mute Regime

8.2 File Checking

One can find 3 useful functions for checking input files in class **File**.

```
static int countBlocks(char* fileName);
```

Count number of data blocks in file with name **fileName**. Data block is a set of lines between empty lines or between empty line and beginning/end of the document. If you don't have empty lines in file then you have only one data block and its number is 1 (not 0). **Important:** line which starts with unprintable character (**space** or **tab**) counts like empty! That is why data lines cannot start with unprintable character.

```
static int countLinesInBlock(char* fileName, int blockNumber = 1);
```

Count how many lines you have in concrete block. If you have only one data block in file you can scip the second argument).

```
static bool checkAllBlocksHaveTheSameSize(char* fileName);
```

Returns **true** if all data blocks in the file have the same number of lines. You might need this function if you work with statistics. So it is important that all polymers which you take for the statistics are of the same size.

8.3 show Number Of Lines In Blocks

The declaration of this function is **void showNumberOfLinesInBlocks(char* fileName)**. It is supposed to print on screen list of blocks in the file and number of lines in each block. This function is created only for user, it means that the program itself never uses it. That is why: **Important:** the function will print on screen even if all verbose are **false**. Do not call this function if you do not want to listen to it.

9 List of Errors

Any errors appears on screen and starts with information about its location. So developer can quickly fined it in the code. After printing the error massage on the screen the program stops with **exit(1)**.

Here is the list of all possible detectable errors:

- **Error in ScalingParam::findEverythingFromS: not enough accuracy for this scaling parameter.** It happens when you want to use scaling parameter with fraction part smaller than **ACCURACY**$^{-1}$ which by default is equal to 0.0001. This restriction is artificial. Read section 4.2 for the solution.
- **Error: cannot open file '<path/fileName.extension>' in <functionName>.** Either you give incorrect name/path of the file or this file doesn't exist at all.
- **Error: cannot create file '<path/fileName.extension>' in <functionName>.** The path doesn't exist.
- **Error in Polymer::readFileWithCoordinates: number of the first block is 1 in data files. You passed me 0!** Make sure that pass the correct number of blocks.
- **Error: void pointer in <functionName>.** This error can happen only if you developing. There is more information can appear in error message.

10 Visualization in MATLAB

There are two MATLAB functions described in two **.m**-files with corresponding names located in directory **PCA/tools**.

10.1 Picture

You can draw on screen any configuration at any step of scaling procedure. For doing this call **showConfiguration('<polymerName>', <number of step>)** In MATLAB's workspace. If you call just **showConfiguration('<polymerName>')**, without the second argument, then you will see the original configuration (i.e step = 0). The program will try to open this file:
"results/<polymerName>_configurations.dat", so be sure that you run scaling procedure and have file with configurations in **/result** directory for corresponding polymer before you call the function.

If you call this function again without closing the figure window, MATLAB will draw the new configuration in the same figure. It is convenient if you want to see how the chain changes during scaling procedure. The colors for atoms and links are picked on random, thus sometimes there could be strange combinations of them.

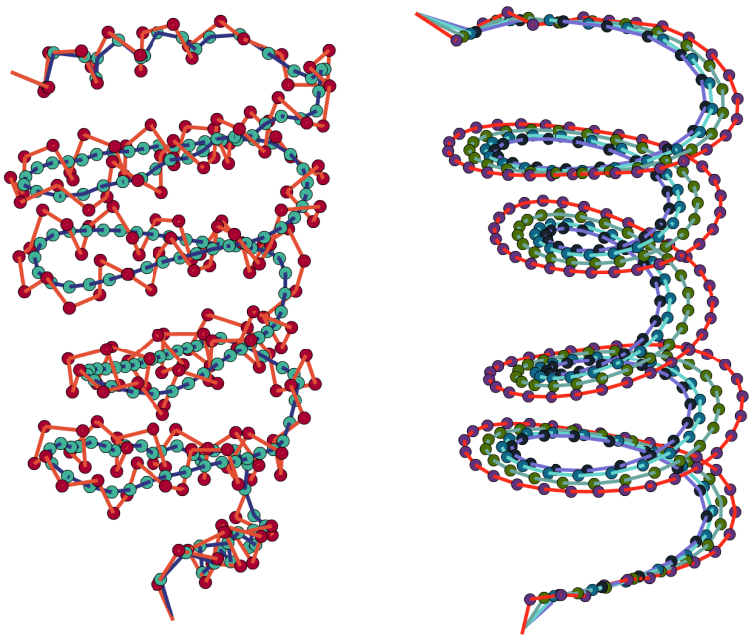


Figure 4: A subfigure

For example, for getting the left figure of Figure 4 one should call:

```
>>showConfiguration('5dn7')
>>showConfiguration('5dn7', 10)
```

And for the right one:

```
>>showConfiguration('5dn7', 50)
>>showConfiguration('5dn7', 80)
>>showConfiguration('5dn7', 100)
>>showConfiguration('5dn7', 110)
```

10.2 Movie

Function **makeScalingMovie('<polymerName>', <increment>)** makes avi-file with scaling procedure where each frame is a chain at some step of rescaling procedure. The first frame is the original configuration. Then will be the step with number = increment. Then 2·increment, 3·increment and so on until only 2 segments left.

If you skip the second argument, then Matlab will record the whole procedure frame by frame.

The same as for the previous function, Matlab will try to open **"results/<polymerName>_configurations.dat"**, so be sure you have it.

10.3 Show Configuration

11 Visualization with Python

In folder **tools** you can find **PlotterPy** - interactive 3D plotter for **.pca** files. The plotter is written in **Python 2.7** with using **Matplotlib**.[4]. There are two ways how you can run the plotters: from command line or from Python shell.

11.0.1 From command line

The program should be run like this:

```
PlotterPy$ python <path/to/>plotter.py <path/to/fileName>
 <configuration1> <configuration2> <...>
```

For example, if you want to open 5dn7.pca which is stored in folder **results**. And this folder **results** is at the same level as folder **PlotterPy**, then you can run from **PlotterPy** folder:

```
PlotterPy$ python ./plotter.py ../results/5dn7.pca
```

and see the original configuration, i.e. configuration number 0.

Try:

```
PlotterPy$ python ./plotter.py ../results/1abs.pca 50 80 100 110
```

to see the corresponding step of rescaling procedure together and get picture similar to 4 at the right.

This method is convenient when you want quickly plot configurations. However, if you want to run this program a lot of times, especially if your files are big, we recommend you to go for option two.

11.0.2 From Python shell

Running from Python shell will require a bit of work from you, but will same your time in a long run. First of all we should launch Python:

```
PlotterPy$ python
```

Then we want to import system library and file with model:

```
>>> import sys
>>> import Polymer
```

Now we have to set polymer from file. If we have the same file organization as in the previous section then we do this:

[4]If you use MacOS, then you do not need to install anything

```
>>> polymer  = Polymer.Polymer('../results/5dn7.pca')
```

The next step is to set configurations we want to plot in the same figure. We want to pass them as arguments. The first argument mast be the name of the program itself:

```
>>> sys.argv = ['plt.py' , '0' ,  '10']
```

Now to plot figure similar to 4 at the left we execute the plotter:

```
>>> execfile('plt.py')
```

If you want to plot some other figure, just repeat two last steps with new values for number of configurations:

```
>>> sys.argv = ['plt.py' , '12' ]
>>> execfile('plt.py')
```

It is also very easy to open another file. To finish Python print exit().
Tip: You can copy-past all the commands from **`plt.py`**, which you can open with any text editor.

11.1 Make Gif

12 GNUplot